FUNDAMENTOS Y APLICACION AUTOMATICA DEL PARALELISMO EN INTELIGENCIA ARTIFICIAL

MANUEL DE HERMENEGILDO SALINAS
Departamento de Inteligencia Artificial de la Facultad de Informática (U.P.M.)

RESUMEN

En este trabajo se revisan conceptos y nociones básicas relacionadas con el paralelismo y se analiza tanto su relación con la Inteligencia Artificial, como las diferentes alternativas de aplicación en el desarrollo de sistemas inteligentes. En primer lugar se argumenta que los procesos típicos de la inteligencia artificial ofrecen posibilidades de explotación de paralelismo, siendo esta explotación una opción muy útil a la hora de incrementar la capacidad y efectividad de los sistemas inteligentes, contrarrestando así el alto coste computacional de los mismos. En segundo lugar se presentan resultados esperanzadores, en el sentido de que la hasta ahora ardua tarea de paralelizar un sistema pueda ser, al menos en parte, realizada automáticamente mediante un análisis en tiempo de compilación del programa que implementa dicho sistema. Para ello se presenta el estudio de un sistema real de paralelización automática de programas lógicos basado en el modelo de paralelismo-And independiente.

1. INTRODUCCION

Existen múltiples razones para que el "paralelismo", entendido como la ejecución simultánea de varias acciones cada una de ellas realizada por diferentes agentes, sea en general, y en el contexto de la inteligencia artificial en particular, un tema de gran interés. Estas razones abarcan desde la similitud con los sistemas inteligentes naturales que, como es evidente en el caso del cerebro, tienen una estructura física paralela, hasta otras más utilitaristas, como es el afán de obtener mejores prestaciones de un sistema inteligente artificial, mediante su implementación sobre una máquina que contenga diversos agentes o procesadores. En el presente trabajo se pretende abordar el tema del paralelismo desde un punto de vista inicialmente general, para profundizar

más tarde en el segundo punto mencionado anteriormente: la mejora de las prestaciones de un sistema inteligente artificial mediante la explotación, idealmente automática, del paralelismo inherente a la tarea inteligente a desarrollar, en un ordenador compuesto por varios procesadores.

En primer lugar, y con objeto de sentar las bases para una discusión posterior, es importante recordar las diferencias entre tres conceptos relacionados: Concurrencia, Multiprogramación y Paralelismo. Las definiciones (informales) que se ofrecen de estos conceptos, y que el lector encontrará familiares, son aquellas que se consideran de general aceptación y más útiles en la práctica.

Informalmente, se entiende por **Concurrencia** la existencia, identificación, o expresión de *múltiples tareas con cierto grado de independencia* en la descripción de un problema. Por ejemplo, en la construcción de una casa la ejecución de cada pared es una tarea con cierto grado de independencia. Al decir esto, expresamos la existencia de *concurrencia* en la construcción de dicha casa. En un ordenador dotado de un sistema operativo sencillo multiusuario, las tareas correspondientes a cada usuario tienen un cierto grado de independencia y, por tanto, puede decirse que existe concurrencia en la ejecución de dichas tareas por parte de dicho sistema operativo. Finalmente, en un cálculo que incluya una multiplicación de un vector por una matriz, las tareas correspondientes a la multiplicación del vector por cada fila o columna de la matriz tienen cierta independencia y podemos, por tanto, decir que hay concurrencia en dicho cálculo.

Se entiende por **Multiprogramación** la ejecución *entrelazada* de varias tareas (concurrentes) por parte de un solo agente o procesador. La ejecución de tareas de modo entrelazado puede tener sentido, entre otras, por razones de eficiencia o para dar una imagen exterior de simultaneidad aparente. Siguiendo con los ejemplos anteriores, una ejecución "en multiprogramación" de la construcción de una casa sería, por ejemplo, aquella en la que un solo operario va poniendo una fila de ladrillos alternativamente en cada pared. Esto puede tener sentido por razones de eficiencia: para que el operario no esté ocioso durante el tiempo de fraguado del cemento correspondiente a una fila de ladrillos. En el caso del sistema operativo multiusuario, la multiprogramación se materializa en que un agente único ejecuta de manera entrelazada las tareas de varios usuarios por motivos tanto de eficiencia (el procesador pasa a ejecutar la tarea de otro usuario mientras la tarea anterior espera, por ejemplo, a acceder al disco), como de simultaneidad aparente (cada usuario recibe atención del ordenador en un tiempo razonable).

Finalmente, se entiende como **Paralelismo** la ejecución *simultánea* de varias tareas, siendo cada una de ellas realizada por un agente o procesador diferente. Retornando a los ejemplos anteriores, en el caso de la construcción de una casa, varios trabajadores pueden trabajar *en paralelo* poniendo cada uno una fila de ladrillos en cada pared, con lo que la casa se construiría más rápido. Es evidente que existe un límite al número de operarios que pueden trabajar simultáneamente en la construcción

de una casa, y que tendrá que haber cierta *comunicación* y *sincronización* entre ellos. En un ordenador con varios procesadores, éstos pueden trabajar en paralelo realizando cada uno la tarea de un usuario. Finalmente, varios procesadores pueden trabajar en paralelo en la multiplicación de un vector por una matriz, realizando cada uno la multiplicación del vector por cada fila o columna de la matriz.

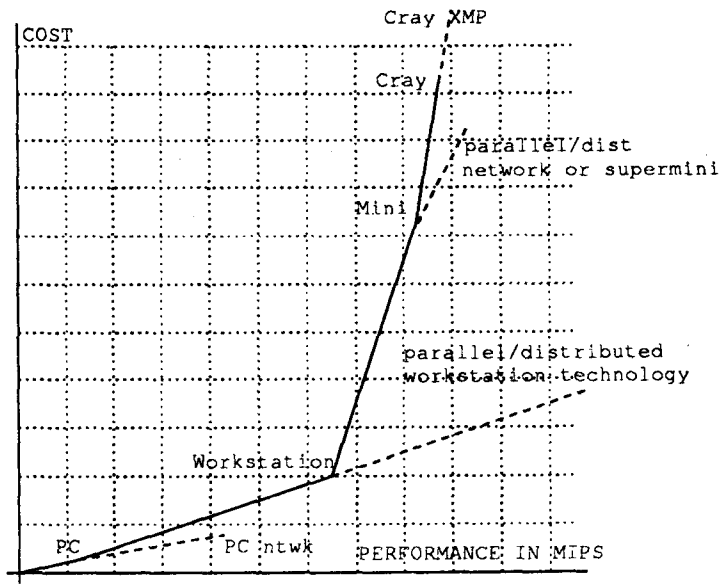

Figura 1: Relación Coste / Prestaciones

De la discusión anterior se desprende que la ejecución en paralelo ofrece ventajas en términos de la velocidad con la que el conjunto de las tareas a realizar puede llevarse a cabo. Aunque quizá de manera menos evidente, el proceso paralelo puede también ofrecer ventajas en términos económicos frente a otras opciones. Ambos tipos de ventajas se hacen especialmente patentes en el diseño de los ordenadores actuales, como se ilustra en la Figura 1, en la que se representa el coste aproximado de una serie de sistemas actuales en función de sus prestaciones. Las líneas continuas representan el coste de sistemas monoprocesador, mientras que las discontinuas representan el coste de sistemas con varios procesadores. Hay varias zonas de interés en la figura. La primera corresponde al segmento en el que se encuentran los sistemas con mayores prestaciones: en este caso el objetivo es obtener el nivel máximo de prestaciones a cualquier coste. Se trata de los supercomputadores, máquinas en la clase de los Cray, en las que en la construcción del procesador la tecnología ha sido llevada al límite. En este caso, para obtener mayores prestaciones es necesario recurrir al paralelismo, es decir, a la inclusión de múltiples procesadores (como en el caso del Cray XMP).

Otra zona muy interesante y que ilustra la utilidad del paralelismo en términos de *economía* corresponde a la de las estaciones de trabajo actuales. Estas máquinas

ofrecen una excelente relación coste/prestaciones (Figura 1). Sin embargo, una vez situados en el segmento superior de esta tecnología, es decir, utilizando el microprocesador de mejor arquitectura, la máxima frecuencia de reloj, memoria caché de gran capacidad y velocidad, entrelazamiento, etc., obtener prestaciones más allá de este punto, manteniendo la arquitectura monoprocesador en el sistema, representa un importante salto económico. Ello es debido a que se hace necesario cambiar a una tecnología generalmente mucho más costosa (por ejemplo de CMOS a ECL, o de ECL a GaAs). Una alternativa clara es la utilización de sistemas con varios procesadores. El paralelismo ofrece en estos casos la posibilidad de un aumento económico de prestaciones, manteniendo la tecnología dada.

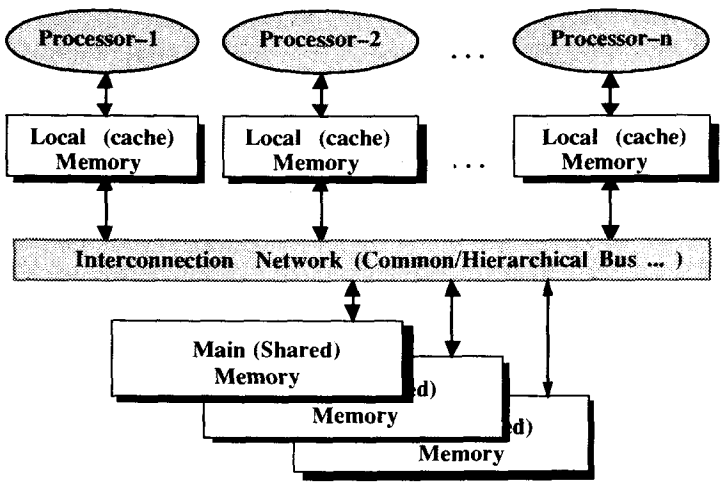

Figura 2: Arquitectura Multiprocesador

Los sistemas con varios procesadores suelen ser de dos tipos: **Fuertemente Acoplados** (también llamados Multiprocesadores) y **Distribuidos** (también llamados Multicomputadores). Una manera de diferenciar estos dos tipos de sistemas sin tener que recurrir a la forma de implementación, es a través de una comparación de las velocidades de acceso a memoria y de comunicación entre los procesadores. Los multiprocesadores son máquinas con varias CPUs que se comunican a velocidad comparable a la de proceso y acceso a memoria. Una implementación habitual de estas arquitecturas es como multiprocesadores de memoria compartida (Figura 2), ejemplos de los cuales son los multiprocesadores Sequent [24] y SUN. Sin embargo, existen numerosas alternativas a este tipo de implementación, como son, por ejemplo, las presentadas en [29]. Las máquinas distribuidas o multicomputadores se caracterizan por tener varias CPUs (pudiendo ser simplemente varias máquinas conectadas) que se comunican a velocidad menor que la de proceso y acceso a memoria. Los procesadores de estas máquinas suelen estar comunicados entre sí por redes de interconexión con topologías de hipercubo o similares y comunicarse mediante mensajes. Ejemplos de máquinas de este tipo son el Intel Hypercube, Transputer, etc.

2. PARALELISMO E INTELIGENCIA ARTIFICIAL

Una vez introducido el concepto básico de paralelismo, sus ventajas en términos de prestaciones y economía, y los tipos principales de máquinas paralelas disponibles en la actualidad, cabe preguntarse sobre la relación entre el Paralelismo y la Inteligencia Artificial.

Se puede empezar observando que en la construcción de sistemas inteligentes "naturales" (por constraste con los artificiales) la naturaleza ya ha hecho amplio uso de la buena relación Prestaciones/Economía que da el paralelismo. El ejemplo más claro es quizá el del cerebro, que consigue solucionar problemas complejos a gran velocidad (en la mayoría de los casos más rápidamente de lo que cualquier supercomputador actual podría conseguir) con elementos de tecnología relativamente "lenta" (las neuronas) pero empleando un gran número de ellos y con un alto grado de paralelismo. A pesar de ser un ejemplo claro de sistema inteligente paralelo natural, la falta de comprensión de los procesos que se llevan a cabo en él, hace del cerebro un ejemplo difícil de analizar más allá del nivel superficial. Un ejemplo quizá más interesante (por ser un sistema mejor comprendido que el cerebro) es el del ojo humano en la tarea de visualizar el entorno. La *concurrencia* de la tarea de visión es a nivel de punto o "pixel" de la imagen, es decir, la *detección* de cada punto de la imagen (su luminosidad, color, etc.) puede hacerse independientemente. La naturaleza ofrece en este caso una interesante combinación de proceso paralelo y "multiprogramación". La retina utiliza *proceso paralelo* al analizar una zona del campo visual (a la que mira el ojo), mediante unos 10^8 receptores que analizan simultáneamente otros tantos puntos de la imagen y codifican la información para su transmisión al cerebro a través de 10^6 fibras. Sin embargo, de esta manera sólo se analiza una parte del entorno. Para visualizar la totalidad del entorno, el ojo ha de efectuar un barrido secuencial del campo visual, fijándose alternativamente en unas u otras zonas del mismo, lo que representa una forma de *"multiprogramación"*. Esta forma de solucionar el problema tiene una importante justificación evolutiva por su excelente relación velocidad/economía y la viabilidad de su realización con componentes y tecnología "asequibles" (muchas células de pequeño tamaño para compensar la relativa lentitud de barrido del ojo). Es interesante comprobar cómo la tecnología utilizada en las cámaras de vídeo ha evolucionado hacia un sistema similar al utilizado por el ojo: mientras una cámara de vídeo tradicional utilizaba un sólo detector (tubo de rayos catódicos) barriendo la imagen rápidamente —un elemento relativamente grande pero capaz de alta velocidad de barrido— las cámaras actuales, de tecnología CCD, se construyen en matrices similares a las de la retina, de elementos mucho más lentos pero de mucho menor tamaño.

La observación de la explotación masiva del paralelismo por parte de los sistemas inteligentes naturales lleva al planteamiento de la utilidad del paralelismo en la implementación de sistemas inteligentes artificiales. Dichos sistemas se han construido hasta ahora sobre monoprocesadores de alta velocidad. Esto sería una opción razonable siempre y cuando las prestaciones de las aplicaciones resultantes fueran sufi-

cientes y el coste del sistema utilizado razonable. Sin embargo, aunque la inteligencia artificial encuentra muchas aplicaciones prácticas siguiendo este método de implementación, existen otras muchas aplicaciones que, o bien requieren sistemas de muy alta velocidad (que resultan muy costosos y podrían ser substituidos por sistemas paralelos construidos con elementos más económicos), o bien requieren prestaciones que los sistemas más rápidos disponibles no son capaces de facilitar (prestaciones que quizá podrían obtenerse con conjuntos de dichas máquinas de las más altas prestaciones trabajando en paralelo). En efecto, la mayoría de las aplicaciones de interés en Inteligencia Artificial son grandes y complejas, con costosos procesos de búsqueda e inferencia, y con mezclas de proceso simbólico, numérico y bases de datos, que con frecuencia se aproximan o superan lo límites de las capacidades de los sistemas actuales. El paralelismo puede ser una opción muy útil a la hora de incrementar la capacidad y efectividad de los sistemas actuales, paliando en cierta medida los problemas mencionados.

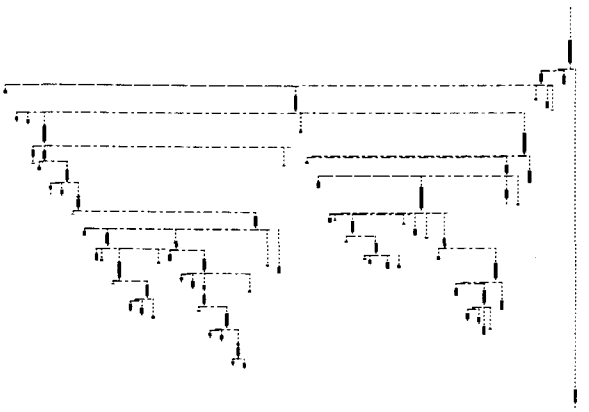

Figura 3: Proceso de búsqueda/deducción

La primera cuestión a resolver es la de si hay concurrencia en los procesos propios de la inteligencia artificial, puesto que si estos procesos fueran estrictamente secuenciales (faltos de concurrencia), es decir, si las tareas a realizar no pudieran ser divididas en subtareas con un cierto nivel de independencia, no sería posible explotar el paralelismo en su implementación. Afortunadamente, muchos de los problemas y algoritmos propios de la inteligencia artificial presentan un alto potencial para la ejecución paralela. Por ejemplo, muchos de los procesos de razonamiento, deducción, inferencia, etc., implican búsquedas por espacios de estados (Figura 3). Las búsquedas por las ramas de estos espacios suelen tener cierta independencia, pudiendo proceder en paralelo. En muchos casos partes de cada rama son también de alguna manera independientes, por lo que pueden ser exploradas en paralelo. Los problemas de visión y análisis del lenguaje humano ofrecen en sus procesos de bajo nivel altos niveles de concurrencia por división espacial y temporal, mientras que los procesos

relacionados de alto nivel son similares a los del ejemplo anterior. Finalmente, los sistemas "neuronales", aquellos que tratan de construir comportamiento inteligente simulando la funcionalidad de los componentes de los sistemas inteligentes naturales (Redes Neuronales, algoritmos biológicos, máquinas de Hoffman, etc.), implican altos niveles de concurrencia, de forma semejante a los sistemas biológicos correspondientes.

Con una cierta convicción de que hay suficientes procesos propios de la inteligencia artificial con un nivel de concurrencia adecuado, la segunda cuestión a resolver sería la de cómo identificar y explotar mediante paralelismo dicha concurrencia. Evidentemente, esto puede hacerse de una manera explícita al programar el sistema inteligente. Sin embargo, la tarea de programar este tipo de sistemas es ya compleja aún antes de preocuparse por la explotación del paralelismo. Por ello, sería deseable que dicho paralelismo se explotara de forma automática en la medida de lo posible. Este tema no es particular de los sistemas propios de la inteligencia artificial. En efecto, aunque hoy en día existen en el mercado muchos tipos de ordenadores paralelos (e incluso parece que los sistemas multiprocesadores serán la norma más que la excepción en un futuro claramento próximo), la cantidad de software que puede explotar la capacidad de rendimiento de estas máquinas es todavía francamente pequeña. Esto se debe en gran parte a lo difícil que, aun hoy en día, resulta para un usuario el extraer el paralelismo inherente a un problema de forma que pueda ser aprovechado realmente por una arquitectura práctica de múltiples procesadores. Aunque los usuarios con cierta experiencia frecuentemente tiene una intuición acertada (en muchos casos mejor que la de cualquier compilador paralelizante) acerca de qué partes del problema (y, correspondientemente, qué partes del programa) pueden ser resueltas en paralelo, la tarea de determinar correctamente las *dependencias* entre dichas partes y la secuencialización y sincronización necesarias para reflejar dichas dependencias, resulta ser muy compleja y dada a errores. Este hecho ha sido mencionado recientemente por Karp [17]:

> "the problem with manual parallelization is that much of the work needed is too hard for people to do. For instance, only compilers can be trusted to do the dependency analysis needed to parallelize programs on shared-memory systems." [2]

Por tanto, un entorno de programación paralela atractivo parece ser uno en el que el programador pueda elegir libremente entre concentrarse únicamente en la tarea de programación convencional (dejando que un compilador paralelizante descubra el paralelismo en el programa resultante) o, alternativamente, en el que pueda hacerse cargo también de la tarea de anotar el programa para explotar el paralelismo. En este último caso, y en vista de la observación de Karp, el compilador debería de ayudar al usuario en las tareas de análisis de dependencias. De hecho, el progreso desde

---

(2) "El problema de la paralelización manual es que la mayor parte del trabajo es demasiado difícil para una persona. Por ejemplo, la labor de realizar el análisis de dependencias necesario para la paralelización de programas para arquitecturas de memoria compartida, sólo puede ser confiada a un compilador."

sistemas que requieren del programador la creación explícita y asignación de procesos a una topología de interconexión particular y el control detallado de la granularidad, a sistemas que no requieren al menos algunas de estas tareas, es un paso adelante en computación paralela, similar al que supuso la aparición del concepto de la memoria virtual (como substituto de los "overlays") o de los lenguajes de programación de alto nivel (como substitutos de la programación en lenguaje máquina). [3]

Karp también menciona la falta de compiladores paralelizantes verdaderamente efectivos y predice que la tecnología correspondiente está todavía a numerosos años vista. Una razón que justifica esta afirmación es que Karp tiene en mente los lenguajes de programación convencionales. Estos lenguajes tienen una semántica imperativa compleja que hace muy difícil la labor del compilador y fuerzan al usuario a utilizar mecanismos de control que esconden o enturbian el paralelismo inherente a un problema. Es por tanto, muy difícil producir algoritmos de paralelización para dichos lenguajes que sean efectivos y cuya corrección se pueda demostrar formalmente.

Por otra parte, los lenguajes de caracter declarativo y, en particular, los lenguajes lógicos, requieren una explicitación del control mucho menor, de tal manera que los programas escritos en dichos leguajes contienen de una manera implícita una parte mucho mayor del paralelismo que pudiera existir originalmente en el problema. Adicionalmente, su semántica los hace comparativamente candidatos mucho más favorables para análisis complejos en tiempo de compilación y, en particular, para la paralelización automática. Finalmente, su base formal (la lógica de predicados para cláusulas de Horn) hace posible, e incluso relativamente sencillo, un tratamiento formal de las tecnicas usadas en el análisis del programa.

De hecho, y como se espera demostrar en el resto del presente trabajo, **la paralelización automática de programas lógicos es ya hoy una realidad,** siendo quizá el paradigma de programación no explícitamente paralelo en el que mayores éxitos se han conseguido hasta la fecha en la explotación automática de sistemas multiprocesadores. Esto es de especial relevancia en el campo de la inteligencia artificial ya que, independientemente de las virtudes desde el punto de vista de la explotación del paralelismo mencionadas anteriormente, en el ámbito de la inteligencia artificial la programación lógica tiene una larga tradición de conveniencia y efectividad en la tarea de implementación de estas aplicaciones, generalmente complejas y con alto componente simbólico. Históricamente, la lógica ya se erigió como substituta ideal del lenguaje natural (ambiguo y necesitado de contexto) en la expresión y formalización del

— — — — — —

(3) Por supuesto, de la misma manera que algunas veces puede haber todavía razones válidas para programar en lenguaje máquina (sobre todo en partes del programa a las que el rendimiento de la aplicación es altamente sensible) puede haber algunos casos en los que la paralelización completamente explícita sea la mejor opción.

razonamiento humano desde los primeros tiempos. La lógica de predicados de primer orden ofrece generalidad, gran poder de expresión en muchas áreas, precisión, flexibilidad y, desde el descubrimiento por Robinson del principio de Resolución [22], la existencia de un método de deducción efectivo que le dota de una capacidad computacional equivalente a la de una máquina de Turing, es decir, capaz de expresar e implementar todas las funciones computables. Aunque la lógica pura presenta ciertas limitaciones en algunos casos (por ejemplo, en presencia de información imprecisa), los sistemas prácticos de programación lógica unen a las ventajas de la lógica (cuando son usados de manera "pura") el permitir *expresar de manera precisa y eficiente* la heurística e imprecisión, implementar lógicas de orden superior, así como otras representaciones (p.e. marcos, redes semánticas) y métodos de inferencia.

El resto del presente trabajo presentará algunos de los avances recientes en el campo de la paralelización automática de programas lógicos y, en particular, en la explotación del llamado *paralelismo-And independiente* mediante el estudio de un sistema real de programación lógica, el sistema &-Prolog, que implementa este tipo de paralelismo.

3. TIPOS DE PARALELISMO EN PROGRAMAS LOGICOS

El paralelismo existente en un programa lógico puede ser, básicamente, de dos tipos [6]: **paralelismo-And** y **paralelismo-Or**. Considérese el siguiente ejemplo para determinar la tripulación de un avión:

tripulacion (X,Y):- navegante (X), piloto (Y)
tripulacion (X,Y):- mecanico (X), piloto (Y)

El significado de la primera cláusula es el siguiente: una tripulación para un avión puede formarse con X e Y, si se puede demostrar que X es navegante e Y es un piloto. En la segunda cláusula se describe otra manera de formar una tripulación en la que X debe ser un *mecanico* en vez de un *navegante.*

Claramente, existen dos caminos *alternativos* que satisfacen la consulta

:- tripulacion (peter, P)

los cuales se corresponden con las dos cláusulas que definen *tripulación.* El paralelismo resultante de la ejecución simultánea de cada uno de los posibles caminos realizada por dos procesadores diferentes, se denomina paralelismo-Or. En general, el paralelismo-Or se corresponde con la ejecución paralela de las diferentes ramas del árbol de demostración, correspondiente a las distintas cláusulas en que se puede expandir una meta determinada.

Más formalmente, y siguiendo la notación y conceptos comunmente aceptados en Lógica Computacional [18], en un punto del proceso de resolución en el que

el resolvente es $G=(g_1, \ldots, g_n)$, la substitución actual θ, el objetivo seleccionado para resolución g_1, y hay dos cláusulas:

$g'_1 \leftarrow g'_{11}, \ldots, g'_{1m}$.
$g''_1 \leftarrow g''_{11}, \ldots, g''_{1m}$.

tales que el unificador más general $mgu(g_1, g'_1)=\theta'$ y $mgu(g_1, g''_1)=\theta''$, se construyen dos resolventes

$G'=(g'_{11}, \ldots, g'_{1m}, g_2, \ldots, g_n)$ con $\theta\theta'$
$G''=(g''_{11}, \ldots, g''_{1m}, g_2, \ldots, g_n)$ con $\theta\theta''$

La ejecución secuencial resuelve primero G' y luego G''. La ejecución en paralelismo-Or supone la ejecución paralela de estos dos resolventes. Como G' y G'' son completamente independientes se obtienen los mismos resultados que en la ejecución secuencial, y todas las ramas se pueden explorar en paralelo.

El paralelismo-Or es característico de los problemas no determinísticos, es decir, problemas de búsqueda. Conceptualmente, este tipo de paralelismo es bastante sencillo, gracias a lo cual su implementación sobre programas lógicos está prácticamente resuelta [27,1].

Si en el ejemplo anterior consideramos la ejecución de una de las cláusulas alternativas, por ejemplo la primera, vemos que para satisfacer *tripulación* sería necesario satisfacer tanto *navegante (peter)* como *piloto (P)*. El paralelismo resultante de la ejecución simultánea de ambas metas realizada por dos procesadores diferentes, se denomina paralelismo-And. En general, el paralelismo-And se corresponde con la ejecución en paralelo de las metas en el cuerpo de una cláusula.

Más formalmente, en un punto del proceso de resolución en el que el resolvente es $G=(g_1, \ldots, g_n)$ y la substitución actual θ, la ejecución secuencial proseguiría resolviendo $g_1\theta$ para obtener θ_1, $g_2\theta\theta_1$ obteniendo θ_2, $g_3\theta\theta_1\theta_2$ obteniendo θ_3, etc., donde la composición de substituciones viene definida por $\forall$ término t, $\theta\eta(t)=\eta(\theta(t))$. La ejecución paralela de g_i y g_j se puede en principio definir como una partición de G en los resolventes $G_1=(g_i)\theta$, $G_2=(g_j)\theta$, y $G_3=(g_1, \ldots, g_i - 1, g_i + 1, \ldots g_j - 1, g_j + 1, \ldots g_n)\theta$, donde luego G_1 y G_2 se resuelven en paralelo obteniéndose θ_1 y θ_2 respectivamente, se aplica la "composición paralela" de θ_1 y θ_2 a G_3, y se resuelve G_3 [14].

El paralelismo-And, por contraste con el -Or, se presenta tanto en problemas determinísticos como no determinísticos, lo que amplía considerablemente su campo de acción. Sin embargo, como veremos a continuación, su implementación es una tarea particularmente delicada debido a las interrelaciones entre las metas que se desean ejecutar en paralelo.

4. PROBLEMATICA DEL PARALELISMO-AND

La ejecución paralela tiene como meta resolver el problema en un tiempo menor o igual al necesario en la ejecución secuencial. Sin embargo, ciertas implementaciones del paralelismo-And pueden llevarnos a un aumento de dicho tiempo de ejecución. Supongamos que se realiza una implementación en la que, durante la ejecución de un programa al invocar una cláusula, se permite que todas las metas en el cuerpo de dicha cláusula se ejecuten en paralelo sin ningún tipo de restricción. Consideremos la siguente cláusula:

piloto (P) :- licencia (P), medico (P)

cuyo significado es el siguiente: P es un *piloto* si P tiene una *licencia* y ha pasado con éxito el exámen *medico.* Supongamos que la consulta inicial es :- *piloto (peter).* La ejecución secuencial ejecutaría primero la meta *licencia (peter)* y a continuación, si tiene éxito, ejecutaría la meta *medico (peter).* La ejecución paralela en la implementación propuesta, ejecutaría las metas *licencia (peter)* y *medico (peter)* en paralelo. Es evidente que, si la ejecución secuencial tuvo éxito, la ejecución paralela también lo tendrá y en menor tiempo.

Supongamos, en cambio, que la consulta es :- *piloto (P).* La ejecución secuencial ejecutaría primero la meta *licencia (P).* Supongamos que, dado el orden en que están los hechos, la variable P se liga a la constante *peter.* A continuación ejecutaría la meta *medico (peter),* lo que como vimos tenía éxito. Sin embargo, la ejecución en paralelo de las metas *licencia (P) y medico (P)* puede dar lugar a resultados inconsistentes si el orden de los hechos hace que el procesador que ejecute *licencia (P)* ligue la variable P a *peter* y el procesador que ejecute *medico (P)* ligue P a cualquier otra constante. Tras la ejecución paralela de ambas metas, al ver que los resultados son inconsistentes, habría que realizar backtracking y volver a ejecutar hasta que no fueran inconsistentes, pudiéndose así superar el tiempo de ejecución secuencial de forma considerable.

Vemos así que los problemas provocados por conflictos en las ligaduras de las variables se deben a la ejecución en paralelo de metas que comparten alguna variable en tiempo de ejecución. Esto puede ocurrir incluso cuando en el texto de la cláusula no aparecen variables compartidas (tiempo de compilación).

Un método de resolver estos problemas de dependencias sería generar todas las posibles soluciones para cada meta y, posteriormente, eliminar las inconsistencias, determinando así el conjunto de soluciones para la cláusula. La principal ventaja de este método es la sencillez de su implementación. Su desventaja fundamental es la ineficiencia, provocada tanto por la gran cantidad de trabajo redundante realizado, como por el alto coste debido a la función que elimina la inconsistencias. Por si fuera poco, esta función podría necesitar una gran cantidad de memoria para el almacenamiento de datos intermedios.

Una de las primeras aportaciones a la solución de este problema fue dada por Conery en [6]. Su método estaba basado en un algoritmo de ordenación capaz de determinar las dependencias entre metas en tiempo de ejecución. Sin embargo, este método disminuía significativamente el rendimiento del sistema debido a su alto coste computacional.

Posteriormente, Chang [5] propuso otro método, con el cual se podían inferir dependencias entre los datos en tiempo de compilación, a partir de cierta información proporcionada por el programador. Precisamente fue esta característica (el ser realizado totalmente en tiempo de compilación), la que supuso su mayor ventaja: coste computacional nulo, y su mayor inconveniente: la información resultante se basaba en un análisis por el caso peor, es decir, en caso de duda se escogía el caso más desfavorable para preservar la corrección del método. Ello provocaba necesariamente una pérdida de cierta cantidad de paralelismo existente en el programa, pérdida que aunque en la actualidad, dado el alto grado de exactitud al que se ha llegado en el análisis, parece razonable, en su tiempo pareció excesiva.

El método propuesto por DeGroot en su esquema de *paralelismo-And restringido* (RAP) [9] aportaba las ventajas de ambos métodos al establecer un compromiso entre la determinación de dependencias entre metas del cuerpo de una cláusula realizada totalmente en tiempo de ejecución (Conery) y la realizada totalmente en tiempo de compilación (Chang). La solución fue generar en tiempo de compilación varios grafos, uno por cada posible modo de activación de la cláusula, en vez de elegir siempre el peor caso. Estos grafos se combinaban produciendo una única *expresión de grafo condicional*. Con ello se conseguía, por una parte, simplificar considerablemente el sistema necesario en tiempo de ejecución y, por otra, ampliar significativamente el número de casos en los que el paralelismo era detectado. Sin embargo, por una parte esta solución seguía adoleciendo de falta de precisión a la hora de determinar las dependencias entre las metas, no aportaba una especificación del procedimiento a seguir en caso de fallo de alguna de las metas ejecutadas en paralelo, ni indicaba qué algoritmos o heurísticas eran adecuadas para generar las expresiones. Por otra, el costo de los tests propuestos limitaba su rendimiento.

En [15, 19, 16] se desarrolló un conjunto de expresiones de grafo condicionales más sencillo y potente, y se proporcionó un semántica procedural para cláusulas lógicas completa con respecto a la de la ejecución secuencial. Esta versión extendida del paralelismo-And restringido se denominó *paralelismo-And independiente.* Asimismo, se propuso una implementación basada en la Máquina Abstracta de Warren (WAM)[28], que representaba el primer esquema en el que se proporcionaba una descripción detallada de los métodos de implementación. Este tipo de paralelismo-And se denomina *independiente*, debido a su principal característica: una vez establecidas las dependencias entre variables, únicamente se permite ejecutar en paralelo aquellas metas que sean independientes, de forma que sus resultados no sean inconsistentes entre sí [14]. Este método asegura que la ejecución paralela se realiza en el mismo o menor tiempo que la ejecución secuencial al preservar la complejidad esperada por el programador, es decir, se realiza el mismo trabajo que en la ejecución secuencial, pero aumentando el rendimiento gracias a la ejecución en paralelo del mayor número de metas posible.

Existe, por contra, otro tipo de paralelismo-And en el que, una vez determinadas las dependencias entre variables, sí se permite la ejecución paralela de las metas dependientes. Los problemas de ligaduras se resuelven permitiendo la comunicación entre las diferentes metas dependientes, siendo el canal de comunicación precisamente la variable que da lugar a su dependencia. Este método ha dado lugar al denominado paralelismo-And *stream.*

La principal desventaja del paralelismo-And stream radica en la gran complejidad existente a la hora de implementar sú backtracking, es decir, en la complejidad de implementar mecanismos de retroceso en presencia de procesos no deterministas. Como resultado, en muchos de los sistemas que utilizan este tipo de paralelismo se ha optado por abandonar el backtracking y, por tanto, las búsquedas no deterministas del tipo *"don't know"*. Es decir, aquellas que permiten encontrar todas las soluciones a un determinado problema. En cambio, se ha adoptado el indeterminismo del tipo *"don't care"* en el que, en el momento en que se encuentra una solución, se sigue adelante sin poder ya volver al punto pasado, perdiendo así una de las características más interesantes y útiles de los programas lógicos. PARLOG, Concurrent Prolog, y GHC [25] son ejemplos de este tipo de lenguajes denominados de *elección irrevocable* ("committed-choice").

El paralelismo-And independiente, en cambio, no sólo mantiene el indeterminismo "don't know" al permitir el backtracking, sino que mantiene totalmente la semántica operacional de Prolog. Es decir, la ejecución de un programa en un sistema basado en este tipo de paralelismo produce el mismo efecto que su ejecución en un sistema secuencial, siendo la única diferencia la mejora obtenida en el rendimiento del programa. Aun cuando esto se logra restringiendo la cantidad de paralelismo que se aprovecha, es decir, restringiendo el número de metas a ejecutar en paralelo a aquellas que son independientes, como veremos la pérdida establecida por esta restricción no será muy significativa.

Existen otras alternativas que, aunque no se tratarán en este trabajo, conviene mencionar. Una de ellas es mantener la semántica "dont'know" permitiendo únicamente el paralelismo-And stream cuando los objetivos son deterministas. Esta solución, utilizada en el modelo Andorra-I [23], restringe sin embargo la cantidad de paralelismo. Recientemente se ha propuesto una solución que combina las ventajas de este método con las del paralelismo-And independiente: el modelo IDIOM [12]. Otra solución es dejar gran parte del control en manos del usuario, como en el Andorra Kernel Language [13], lo que permite combinar características de lenguajes como Prolog y GHC. Las técnicas utilizadas en el paralelismo independiente combinadas con otros tipos de análisis sirven para salvar la distancia entre la programación lógica tradicional y este tipo de lenguajes, como se demuestra en [3]. Finalmente, es importante señalar otros modelos propuestos que combinan el paralelismo-And con el paralelismo-Or [11, 10] a los que también son aplicables los resultados que se presentan.

4.1. Paralelismo-And independiente estricto.

Tradicionalmente, en lo que se ha denominado paralelismo-And independiente *estricto,* dos metas se consideraban independientes con respecto a una determinada substitución, si y sólo si tras aplicar dicha substitución a cada una de las metas, no compartían variables. Definámoslo formalmente [14]:

Deninición 1 (variables de una meta): *Dada una meta m, denominaremos vars(m) al conjunto de todas las variables que aparecen en m.*□

Definición 2 (independencia estricta): *Dos metas* m_1 y m_2 *son estrictamente independientes con respecto a una determinada substitución* θ*, si y sólo si vars* $(m_1\theta) \cap$ *vars* $(m_2\theta) = \emptyset$. *N metas* $m_1, \ldots, m_n$ *son estrictamente independientes con respecto a una determinada substitución* θ*, si cada pareja de metas son estrictamente independientes con respecto a* θ. *Generalizando, dos metas son estrictamente independientes si son estrictamente independientes con respecto a cualquier substitución* θ. *La extensión a un conjunto de metas es análoga.*□

Es importante tener en cuenta que la definición anterior considera las metas después de haberse aplicado la substitución. Por ello m_1 y m_2 pueden no tener variables en común y, al mismo tiempo, no ser estrictamente independientes para una determinada θ. Bastaría considerar el ejemplo *tripulación* propuesto en 3, en el que las metas *navegante (X)* y *piloto (Y)* no tienen variables en común y, sin embargo, no son estrictamente independientes con respecto a la substitución $\theta = \{X/Z, Y/Z\}$, ya que *navegante* $(X)\theta$ = *navegante (Z) y piloto* $(Y)\theta$ = *piloto (Z),* compartiendo por tanto la variable Z.

Es evidente que una meta básica será estrictamente independiente de cualquier otra meta, ya que su conjunto de variables es vacío y, por tanto, su intersección con cualquier conjunto de variables será también vacío. También es importante observar que la independencia estricta no es transitiva, es decir, aun cuando sepamos que dos metas m_1 y m_2 son estrictamente independientes y que m_2 y m_3 son también estrictamente independientes, no podemos asegurar que m_1 y m_3 sean estrictamente independientes. Consideremos por ejemplo las metas p(X), q(Y) y r(Z) y la substitución $\theta = \{X/f(W), Y/a, Z/g(W)\}$. Es evidente que tanto p(X) y q(Y) como q(Y) y r(Z), son estrictamente independientes con respecto a θ. Sin embargo, $p(X)\theta = p(f(W))$ y $r(Z)\theta = r(g(W))$ comparten W, por lo que no son estrictamente independientes.

La independencia estricta, por tanto, mejora el rendimiento de un programa al permitir ejecutar en paralelo todas aquellas metas en el cuerpo de una cláusula que sean estrictamente independientes, asegurando que los resultados obtenidos mediante esta ejecución son los mismos que los obtenidos en ejecución secuencial y que la cantidad de trabajo realizado no es mayor, como se demuestra en [14].

4.2. Paralelismo-And independiente no estricto

Recientemente, este concepto ha sido ampliado [14], al considerar independientes un conjunto de metas, incluso cuando comparten variables, siempre y cuando cumplan ciertas restricciones. Veámoslo formalmente.

Definición 3 (v- y nv- ligadura): *Una ligadura X/T se denominará v-ligadura si T es una variable. En caso contrario, se denominará nv-ligadura.*□

Definición 4 (independencia no estricta): *Sea $m_1, \ldots, m_n$ un conjunto de metas y θ una determinada substitución. Asimismo, sea $SH = \{V \mid \exists i, j, 1 \leq i, j \leq n, i \neq j, V \in (vars(m_i\theta) \cap vars(m_j\theta))\}$ el conjunto de variables compartidas entre dichas metas con respecto a θ y $M(V) = \{m_i \mid V \in vars(m_i\theta), V \in SH\}$ el conjunto de metas que contienen a cada variable compartida V. Sea θ_i la substitución de respuesta para $m_i\theta$. El conjunto anterior de metas es no estrictamente independiente con respecto a θ si se satisfacen las siguientes condiciones:*

- *Para toda variable $V \in SH$, la única $m_i \in M(V)$ que realiza una nv-ligadura sobre V es la situada más a la derecha;*

- *$\forall_i = 1, \ldots, n$, si vars $(m_i\theta)$ contiene más de una variable compartida, por ejemplo $X_1, \ldots, X_k$, entonces $X_1\theta_i, \ldots, X_k\theta_i$ son estrictamente independientes.*□

Intuitivamente, la primera condición establece que sólo una meta ligue una variable compartida, con lo cual se evitan los problemas de inconsistencias por dependencias. La condición de que esta meta sea la situada lo más a la derecha se introduce para evitar que la ejecución paralela realice más trabajo que la secuencial. Es decir, evitar que existan metas situadas a la derecha de la meta que liga la variable que, en ejecución secuencial, estarían más instanciadas, siendo por tanto su árbol de búsqueda menor que si se ejecutan en paralelo, antes de haber sido instanciada la variable compartida. La segunda condición elimina la posibilidad de introducir nuevas dependencias entre variables compartidas, como resultado de la ejecución de una de las metas que se ejecutan en paralelo.

Es evidente que si un conjunto de metas es estrictamente independiente con respecto a una substitución, entonces es no estrictamente independiente con respecto a esa substitución: cualquier conjunto de metas estrictamente independiente con respecto a una substitución satisface las dos condiciones anteriores, ya que éstas se refieren a variables compartidas (tras la aplicación de la substitución) y, como hemos visto, las metas estrictamente independientes no comparten variables (tras ser aplicada la substitución).

Este tipo de independencia da lugar al paralelismo-And independiente *no estricto*, en contraposición al paralelismo-And independiente estricto descrito anteriormente. Como se ha determinado en [26], este tipo de paralelismo puede ser bastante

frecuente, y por tanto su utilización permite mejorar el rendimiento de los programas de forma considerable.

5. EL LENGUAJE &-PROLOG

El lenguaje &-Prolog fue desarrollado con el fin de servir de medio para expresar e implementar el paralelismo-And independiente. Es, esencialmente, una extensión del lenguaje Prolog obtenida mediante la adición de un *operador de paralelismo* "&" y un conjunto de predicados predefinidos relacionados con el paralelismo. Sus principales características son las siguientes:

- El operador de ejecución paralela "&" se utiliza para indicar las metas que pueden ejecutarse en paralelo, es decir a, b indica que a y b se ejecutan secuencialmente en dicho orden, a & b indica que a y b pueden ser ejecutadas en paralelo.

- Existen dos tipos de tests realizados en tiempo de ejecución que aseguran la independencia de las metas: tests de basicidad y tests de independencia sobre las instanciaciones de las variables en tiempo de ejecución.

- Contiene primitivas y predicados predefinidos que permiten expresar no sólo paralelismo-And estricto, sino grafos de ejecución no estricta e incluso paralelismo-And dependiente [19].

Considérese de nuevo la cláusula piloto:

piloto (P) :- licencia (P), medico (P)

Como se señaló anteriormente, antes de permitir la ejecución paralela de las metas, es necesario comprobar que sus instanciaciones en tiempo de ejecución sean independientes. Como P es una variable común a las metas *licencia (P) y medico (P),* la única forma posible de que exista independencia estricta entre ambas metas es que P esté ligado a un término *básico,* es decir, a un término que no contenga variables. Para expresar esto, la cláusula anterior se anotaría en &-Prolog de la siguiente forma:

piloto (P) :- ground (P) → licencia (P) & medico (P)
; licencia (P), medico (P)

Donde "→" y ";" se utilizan como construcción "if→then ; else", de la misma forma que en Prolog. El significado operacional de la cláusula es el siguiente: "si P está ligada a un término básico en tiempo de ejecución, ejecutar las metas *licencia (P) y medico (P)* en paralelo; si no, ejecutarlas secuencialmente en orden de izquierda a derecha, como en Prolog". El test de basicidad, *ground (P),* se satisfará, si y sólo si en tiempo de ejecución P está instanciada a un término básico.

Veamos ahora la cláusula *tripulacion:*

tripulacion (X,Y) :- navegante (X), piloto (Y)

Para poder ejecutar en paralelo las dos metas del cuerpo de la cláusula las instanciaciones en tiempo de ejecución de X e Y han de ser términos independientes. Para expresar esto, la cláusula anterior se anotaría en &-Prolog de la siguiente forma:

tripulacion (X,Y):-
indep (X,Y) → navegante (X) & piloto (Y)
; navegante (X) , piloto (Y)

El test de independencia indep (X,Y) se satisfará si y sólo si, en tiempo de ejecución, X e Y están instanciadas a términos que no comparten variables. Estos tests se definen formalmente como [14]:

Definición 5 (ground) *ground (X)* $\equiv vars\ (term_X) = \emptyset$, *donde* $term_X$ *es la instanciación en tiempo de ejecución de X.*□

Definición 6 (indep) *indep (X,Y)* $\equiv$ vars $(term_X) \cap vars(term_Y) = \emptyset$, *donde* $term_X$ *y* $term_Y$ *son las instanciaciones en tiempo de ejecución de X e Y respectivamente.*□

Las expresiones condicionales en los anteriores ejemplos (y una clase considerable de las expresiones útiles en general) pueden ser generalizadas a:

$$(i\_cond \rightarrow meta_1\ \&\ meta_2\ \&\ \ldots\ \&\ meta_N\ ;\ meta_1\ ,\ meta_2\ ,\ \ldots\ ,\ meta_N)$$

donde i_cond es la conjunción de tests de basicidad e independencia sobre las variables que aparecen en las metas, pudiendo ser éstas simples literales o expresiones complejas sobre literales. Por conveniencia sintáctica, estas expresiones se abrevian como **Expresiones de Grafo Condicional (CGE):**

$$(i\_cond \rightarrow meta_1\ \&\ meta_2\ \&\ \ldots\ \&\ meta_N)$$

6. EL SISTEMA &-PROLOG

La figura 4 muestra la estructura conceptual del sistema &-Prolog. Aunque en la figura se representan los diferentes componentes del compilador como módulos, en realidad el conjunto es una unidad integrada que ofrece un entorno completo de programación paralela basado en el lenguaje Prolog. El sistema ofrece como subsistema una implementación completa del lenguaje Prolog compatible tanto desde el

punto de vista sintáctico como semántico con el estándard DECsystem-20 Prolog/Quintus Prolog, también llamado estándard de Edimburgo, que es el más reconocido hoy en día.

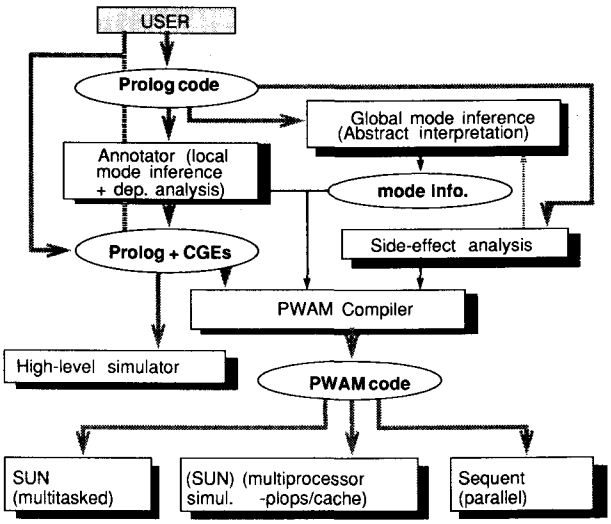

Figura 4: Arquitectura del Sistema &-Prolog

La interfase con el usuario es la normal, en la que se proporciona un bucle interactivo en el que el usuario puede hacer preguntas y tiene acceso al intérprete y al compilador. En este bucle, el usuario puede interpretar y compilar código Prolog "convencional" (es decir, no paralelizado). En este modo de funcionamiento el usuario no es consciente (excepto por la mejora en el rendimiento, es decir, la reducción del tiempo de ejecución) de que el sistema es diferente en modo alguno de un sistema Prolog habitual. Por supuesto, a través de directivas al compilador es posible desactivar la paralelización automática. El usuario puede proporcionar también código ya paralelizado (código &-Prolog) en cuyo caso el compilador respetará las anotaciones dadas. Esta paralelización manual selectiva puede hacerse a nivel de un fichero completo, una relación o una cláusula, mientras que el resto del programa se continúa paralelizando automáticamente. En cualquier caso, el compilador comprueba la corrección de las anotaciones del ususario.

6.1 Estructura del Compilador

En el compilador, el código del usuario es analizado por cuatro módulos diferentes:

- El **Anotador**, o "paralelizador", es el módulo encargado de detectar oportunidades para la ejecución en paralelo. Para ello, este módulo es capaz de llevar a cabo un análisis de dependencias local del código presentado al sistema. Si, además, se ha seleccionado la opción de análisis global, el anotador recibe información del analizador global (que se describe en el siguiente punto) sobre ciertas características de independencia y carencia de variables libres de los términos a los que se ligarán las variables del programa en tiempo de ejecución.

  El anotador recibe, asimismo, información por parte del módulo de análisis de efectos colaterales (que será también descrito posteriormente) acerca de qué predicados contienen o llaman a algún predicado del sistema no lógico que pueda dar lugar a dichos efectos, como son, por ejemplo, los predicados de lectura y escritura y los que modifican la base de datos.

  Algunos de los algoritmos y técnicas heurísticas usadas en este módulo se describen en [20]. Además, este documento trata el tema de la preservación del paralelismo cuando un grafo de ejecución ideal se convierte en una expresión lineal. Los principios básicos y las propiedades del paralelismo-And independiente en que está basado el proceso de anotación se tratan en [14].

- El **Analizador Global**, o "interprete abstracto", analiza el programa ejecutándolo sobre un dominio abstracto [7]. Este dominio se diseña específicamente con la idea de inferir, con alta precisión, características de independencia de los términos que aparecerán en tiempo de ejecución. El analizador infiere información sobre las posibles substituciones que pueden ocurrir en todos los puntos del programa. Dicha información se envía también al compilador de bajo nivel. Algunos de los conceptos y algoritmos utilizados en el analizador global se presentan en [21, 8]. Algunos de los resultados de los experimentos en la implementación y utilización de la interpretación abstracta se presentan en [30, 8]. Como se demuestra en [8] el número de tests por Expresión de Grafo Condicional se reduce significativamente si se aplica el análisis global, y además, también se generan muchas más Expresiones de Grafo *no* Condicionales (es decir, sin ningún test) cuya ejecución paralela es muy eficiente.

- El **Analizador de Efectos Colaterales** clasifica los predicados del programa como "puros" o "impuros", es decir, que contienen o llaman a algún predicado no lógico y que pueden producir efectos colaterales. El anotador utiliza esta información para introducir, si es necesario, semáforos en las cláusulas con el objeto de garantizar un comportamiento observable idéntico al del lenguaje Prolog, si el usuario así lo desea. Las técnicas utilizadas hasta la fecha en la secuencialización y sincronización a nivel del lenguaje &-Prolog y a nivel de la máquina abstracta se presentan en [19].

- El **Compilador PWAM** efectúa la última etapa del proceso de compilación: la traducción del programa anotado (en el lenguaje &-Prolog) a un código intermedio, correspondiente al lenguaje máquina de una máquina abstracta paralela ideal, la

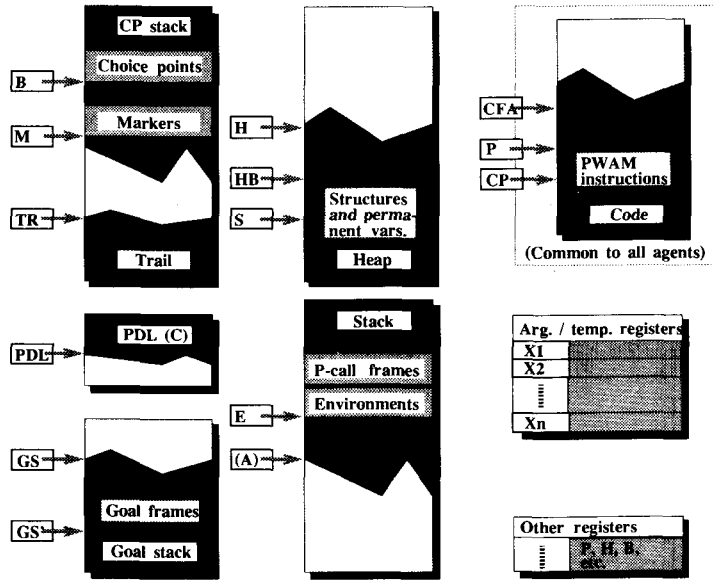

Figura 5: Modelo PWAM: Conjunto de pilas de un Agente

máquina PWAM [15]. Esta máquina abstracta representa una evolución hacia la ejecución paralela de la Máquina Abstracta de Warren (WAM) y se representa esquemáticamente en la figura 5.

Todas las partes y pasos del compilador están escritas en el lenguaje Prolog y, como se comentó anteriormente, están a disposición del usuario "on-line".

6.2 El Sistema de Ejecución Sobre Multiprocesadores

Se ha realizado una implementación del sistema &-Prolog que corre en máquinas paralelas de memoria compartida (como los sistemas Sequent o Encore) y estaciones de trabajo secuenciales (como las SUN). Este sistema, que es una evolución del sistema SICStus Prolog, versión 0.5 [2] implementa la máquina abstracta PWAM a través de un intérprete del código máquina PWAM (en la manera habitual en las implementaciones Prolog y Lisp modernas) escrito en el lenguaje C ampliado con una serie de funciones para acceder directamente a la memoria compartida y a los elementos de sincronización (semáforos). La ventaja de escribir el intérprete en C es su portabilidad. El coste es una cierta pérdida de velocidad con respecto a algunas implementaciones comerciales que están escritas en lenguaje máquina como Quintus Prolog, pero se entiende que esta pérdida es tolerable dado que se trata de un sistema experimental, y que es pequeña. Esto se puede comprobar

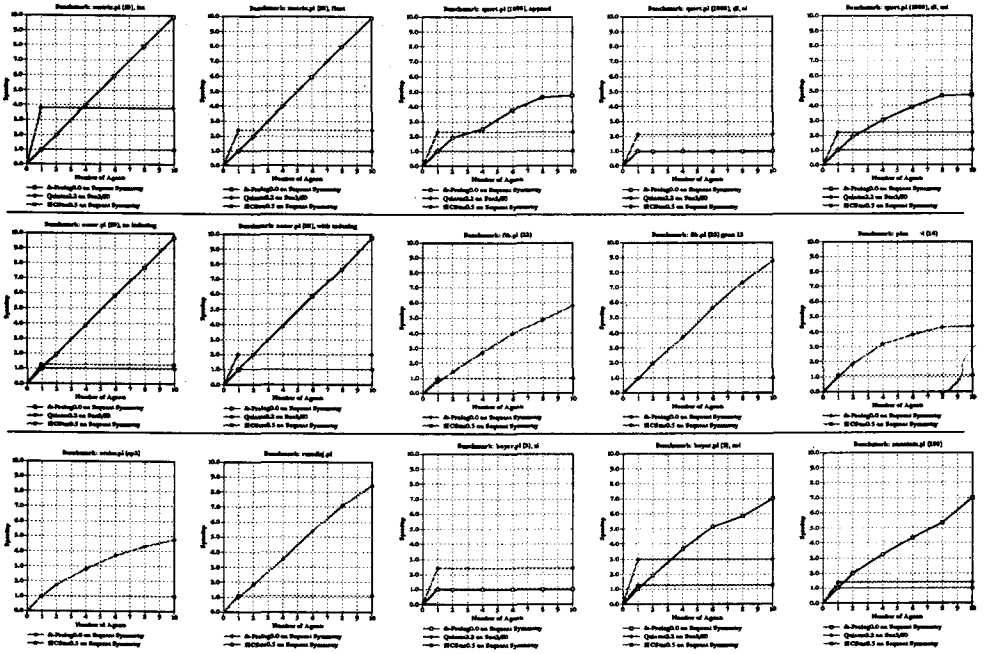

Figura 6: Algunos Resultados del Sistema &-Prolog

en la figura 6. En esta figura se representan los resultados de la ejecución paralela sobre el sistema &-Prolog en un Sequent Symmetry de una serie de programas Prolog paralelizados. En el eje horizontal se representa el número de procesadores y en el vertical la velocidad relativa obtenida con respecto a la velocidad de Sicstus Prolog corriendo el programa secuencial sobre un procesador (línea horizontal inferior). La línea horizontal superior representa la velocidad de Quintus Prolog, una de las implementaciones comerciales de Prolog, de la más alta velocidad. La diferencia entre la velocidad de Sicstus (y de &-Prolog sobre un procesador, de la que apenas difiere) y Quintus, se debe a la diferencia en el lenguaje de implementación (ensamblador frente a C). En cualquier caso, una vez que se utiliza más de un procesador, en la mayoría de los casos &-Prolog obtiene una velocidad superior a la de Quintus Prolog. Aunque los resultados no son siempre tan buenos y, evidentemente, aún queda bastante camino por recorrer en la paralelización automática de programas lógicos (se puede apreciar en la figura 6 que en algunos casos el compilador no es capaz de paralelizar adecuadamente el programa) los resultados son ciertamente prometedores y justifican la continuación de los trabajos en esta línea.

Como ilustración de la ejecución paralela se incluyen algunas figuras realizadas con la herramienta de visualización *visandor* [4] a partir de trazas de

ejecución producidas por el sistema &-Prolog. Veamos, con un sencillo ejemplo, la función de la herramienta. Consideremos el siguiente programa:

```
p :- q, r, s, t
```

paralelizado como:

```
p :- (q & r & s), t
```

Es evidente que todas las metas podrían paralelizarse ya que son independientes, sin embargo hemos querido hacerlo así para obtener una traza más sencilla y completa.

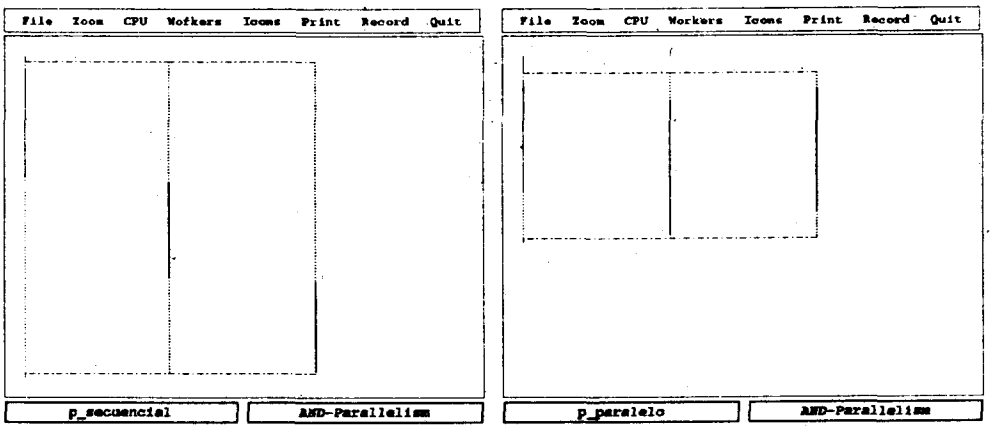

Figura 7: Traza de ejecución secuencial y paralela

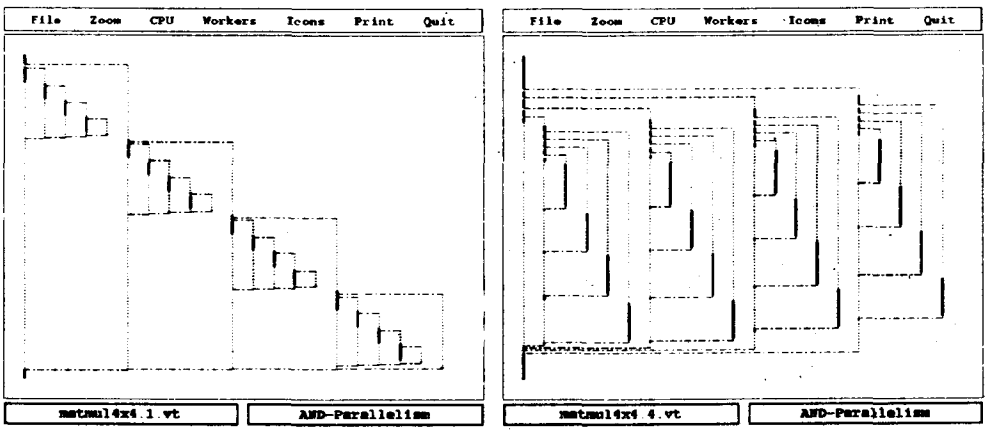

Figura 8: Traza secuencial y paralela: multiplicación 2 matrices 4x4

La visualización de la traza de la ejecución con un único procesador se muestra a la izquierda en la figura 7. En dicha figura, como en las otras generadas con visandor, el tiempo de ejecución avanza verticalmente, las líneas discontínuas significan espera y las líneas contínuas significan ejecución. En la traza se ve claramente la ejecución de tres metas (representadas por las tres líneas verticales contínuas) por parte de un único agente. Este agente ejecuta las metas secuencialmente siguiendo un orden de izquierda a derecha. La parte derecha de la figura muestra la ejecución real realizada en el Sequent con tres procesadores. El tamaño de la figura indica la diferencia de tiempo de ejecución con respecto a la traza anterior (figura 7), diferencia que marca la aceleración conseguida gracias a la paralelización. También se aprecia en la figura el tiempo empleado por el Sequent en la creación de cada uno de los agentes encargados de la ejecución. La figura 8 presenta la visualización de la ejecución secuencial y paralela sobre cuatro procesadores de una multiplicación de dos matrices de 4x4 elementos. En este caso en la figura de la derecha el tiempo se ha "estirado" para ofrecer mejor detalle. La aceleración real es próxima a 4 para dicho número de procesadores.

7. CONCLUSIONES

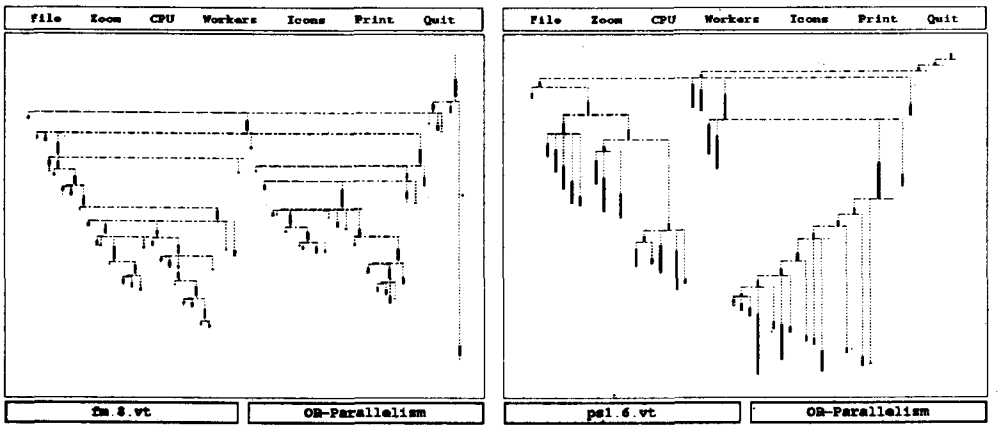

Figura 9: Paralelismo-Or fm, 8 procesadores, psl 6 procesadores

Los resultados presentados en el presente trabajo son alentadores en el sentido de que apuntan a que la paralelización automática de programas escritos en lenguajes usados comunmente en inteligencia artificial, como son los lenguajes lógicos, utilizando paralelismo-And es factible. Se han obtenido resultados también esperanzadores con el paralelismo-Or (la figura 9 presenta una representación, realizada también con la herramienta visandor, de la ejecución en paralelo en el sistema MUSE [1] de dos ejemplos que contienen procesos de búsqueda utilizando paralelismo-Or) y se están desarrollando en la actualidad sistemas que combinan el paralelismo-Or y el -And [11, 10, 12, 13].

Una de las ventajas de la paralelización en el caso de estos lenguajes, es que la velocidad resultante puede superar incluso a la de sistemas escritos en lenguajes de nivel mucho más bajo, como el "C". De esta manera se abre la posibilidad, a través del paralelismo, y gracias al nivel superior del entorno y lenguaje de programación, de desarrollar sistemas en un tiempo menor, con menores errores y esfuerzo, y sin penalización (o incluso ventaja) en la velocidad final del sistema inteligente desarrollado.

BIBLIOGRAFIA

[1] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Perfomance. In *1990 North American Conference on Logic Programming.* MIT Press, October 1990.

[2] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[3] Francisco J. Bueno Carrillo. Traducción Automática de Prolog al Andorra Kernel Language (Automatic Translation from Prolog to the Andorra Kernel Language). Master's thesis, University of Madrid (UPM), Facultad de Informática, Madrid, 28660, 1992.

[4] M. Carro, L. Gómez, and M. Hermenegildo. VISANDOR: A Tool Visualizing And-/ Or-parallelism in Logic Programs. Technical report, U. of Madrid (UPM) Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, 1991.

[5] J.-H. Chang, A.M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218-225, February, 1985.

[6] J.S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin of Programming Languages,* pages 238-252, 1977.

[8] M. J. García de la Banda. Implementación de un Intérprete Abstracto de Programas Prolog sobre el Dominio "sharing + freeness" (Implementation and Evaluation of an Abstract Interpreter over the "sharing + freeness" Domain). Master's thesis, University of Madrid (UPM), Facultad de Informática, Madrid, 28660, 1992.

[9] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems,* pages 471-478. Tokyo, November 1984.

[10] G. Gupta and M. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP'91 Workshop on Parallel Execution of Logic Programs.* Springer-Verlag, June 1991.

[11] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming,* pages 332-349. MIT Press, October 1989.

[12] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism. In *1991 International Logic Programming Symposium.* MIT Press, October 1991.

[13] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming.* MIT Press, June 1990.

[14] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming,* pages 237-252. MIT Press, June 1990.

[15] M. V. Hermenegildo. *An Abstrat Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[16] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third Int'l. Conf. on Logic Programming,* number 225 in LNCS, pages 40-55. Springer-Verlag, July 1986.

[17] A.H. Karp and R.C. Babb. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, September 1988.

[18] J. W. Lloyd. *Logic Programming.* Springer-Verlag, 1984.

[19] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming.* MIT Press, June 1989.

[20] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming,* pages 221-237. MIT Press, June 1990.

[21] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming.* MIT Press, June 1991.

[22] J.A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM,* 12(23):23-41, January 1965.

[23] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming,* pages 443-456. MIT Press, June 1991.

[24] Sequent Computer Systems, Inc. *Balance 8000/21000 Technical Summary,* 1986.

[25] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge MA, 1987.

[26] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium.* MIT Press, October 1991.

[27] P. Szeredi. Perfomance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[28] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[29] D.H.D. Warren and S. Hairidi. Data Diffusion Machine - a scalable shared virtual memory multiprocessor. In *1988 International Conference on Fifth Generation Computer Systems.* ICOT, ICOT, February 1988.

[30] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming.* MIT Press, August 1988.